إهداء الى كل مسلم فى أى مكان

ابدأ LINQ

كيفية العمل بالتقنية الجديدة للإستعلامات باستخدام لغة C#



إبدأ

LINQ

كيفية العمل بالتقنية الجديدة للإستعلامات باستخدام لغة #C

Linverno

2008

بسم الله الرحمن الرحيم

قُلْ إِن كَانَ ءَابَآؤُكُمْ وَأَبْنَآؤُكُمْ وَإِخْوَٰنُكُمْ وَأَزْوَٰجُكُمْ وَعَشِيرَتُكُمْ وَأَمْوَٰلٌ
ٱقْتَرَفْتُمُوهَا وَتِجَٰرَةٌ تَخْشَوْنَ كَسَادَهَا وَمَسَٰكِنُ تَرْضَوْنَهَآ أَحَبَّ إِلَيْكُم
مِّنَ ٱللَّهِ وَرَسُولِهِۦ وَجِهَادٍ فِى سَبِيلِهِۦ فَتَرَبَّصُوا۟ حَتَّىٰ يَأْتِىَ ٱللَّهُ بِأَمْرِهِۦ ۗ
وَٱللَّهُ لَا يَهْدِى ٱلْقَوْمَ ٱلْفَٰسِقِينَ ﴿٢٤﴾

صدق الله العظيم

سورة التوبة

مقدمة

هذا الكتاب جزء من مجهود لخدمة الإسلام والمسلمين. لذلك فهو مجانى, ينتفع به من أراد الإنتفاع. وغير محدد أو مقصور على فئة خاصة. يمكنك ايضاً نسخه و توزيعه و طبعه والإقتباس منه بأى صورة كانت. بل من الأفضل أن يتم توزيعه و نشره قدر الإستطاعه. واتمنى لكل من يستطيع أن يقدم كتب علمية نافعة أن لا يتأخر عن هذا. فخيركم - كما تعلمون – من تعلم العلم و علمه.

هذا الكتاب ليس للمبتدأين فى البرمجة. لابد من دراسة البرمجة الشيئية أولاً ثم دراسة DotNet وبعض من SQL وأساسيات قواعد البيانات, و معرفة القليل عن XML. و الأقل عن ADO.NET. حتى تتمكن من استيعاب هذا الكتاب استيعاب جيد. فهو يقدم تقنية جديدة فى الإستعلام عن البيانات من مصادر البيانات المختلفة. ويقدم شرح لمفاهيم و استخدام لمميزات لغات الدوت نت الحديث (C# 3.0)

هذا الكتاب يحتوى على 50 صفحة تقريباً. لقد راعينا ألا يكون هناك أى زيادات فى الشرح تؤدى الى فقد التركيز أو الإبتعاد عن موضوع الكتاب. لذلك جاء الكتاب مركز و شبه شامل لأساسيات التقنية. هناك العديد من الكتب التى تزيد صفحاتها عن 500 صفحة. لكن فى الحقيقة يمكن إختصارها الى ربع هذا العدد. لذلك ابتعدنا عن كل ما ليس له داعى وقدمنا علماً خالصاً بدون تعقيدات. سوف تجد ان هذا الكتاب يتحدث عن الجديد فى C# و LINQ To SQL فقط. فى المستقبل القريب جداً – إن شاء الله – سوف نقدم كتاب أخر يتحدث عن LINQ To XML.

نرجو من الله أن يكون هذا الكتاب نفعاً لنا فى الدنيا و الأخرة. ونفعاً لجميع المسلمين.

المحتويات

الفصل الأول

ميزات لغة #C

الفصل الثانى

اساسيات LINQ

الفصل الثالث

LINQ To SQL

1

الفصل الأول

ميزات لغة C#

يقدم هذه الفصل الميزات الجديدة فى لغة C#3.0, و ايضاً بعض الميزات فى C# 2.0 و C# 1.0.

المعرفة الكاملة بالتحسينات التى ادخلت على لغة C# فى الإصدار الثالث لها ليست ضرورية لإستخدام LINQ. مع ذلك, سوف نقدم وصف قصير لميزات C# (ابتداءاً من C# 1.x حتى C# 3.0) والتى سوف تحتاجها كى تفهم بوضوح كيفية العمل مع LINQ باقصى كفاءة. إذا قررت أن تتخطى قراءة هذا الجزء, يمكنك ان تعود مرة أخرى فى أى وقت عندما تحتاج الى فهم ما الذى يحدث بالضبط بداخل جُمل LINQ.

فى هذا القسم, سوف نقوم بتوضيح بعض ميزات C# المهمة بالنسبة لـ LINQ: generics, anonymous methods, yield, IEnumerable interface. لابد من أن تفهم هذه المفاهيم كى تستطيع فهم LINQ.

C# 2.0

Generics

إذ اردنا عمل دالة تقوم بطباعة عناصر array من النوع int, فهذا أمر بسيط. انظر الكود التالى:

```
Static void PrintArray( int[] inputArray)
{
      foreach ( int element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}
```

وفى نفس البرنامج إذا اردنا دالة أخرى تقوم بطباعة array من النوع double, فهذا امر أبسط. نكتب نفس الدالة السابقة بنفس الأسم مع تغير نوع البيانات فى عناصر الدالة. كالتالى:

```
Static void PrintArray( double[] inputArray)
{
      foreach ( double element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}
```

وإذا اردنا دالة أخرى تقوم بنفس العملية على اى نوع بيانات أخر, فنحن نقوم بعمل دالة جديدة لكل نوع.
أنظر الكود التالى, هذا الكود لبرنامج كامل:

```
using System;
class OverloadedMethods
{
    static void Main( string[] args )
    {
       int[] intArray = { 1, 2, 3, 4, 5, 6 };
       double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
       char[] charArray = { 'H', 'E', 'L', 'L', 'O' };

       Console.WriteLine( "Array intArray contains:" );
       ProntArray( intArray );
       Console.WriteLine( "Array doubleArray contains:" );
       ProntArray( doubleArray );
       Console.WriteLine( "Array charArray contains:" );
       ProntArray( charArray );
    }

    Static void PrintArray( int[] inputArray)
{
      foreach ( int element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}

Static void PrintArray( double[] inputArray)
{
      foreach ( double element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}

Static void PrintArray( char[] inputArray)
{
      foreach ( char element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}
}
```

فى البرنامج السابق قمنا بعمل ثلاثة دوال تقوم بنفس العمل مع تغير نوع العناصر التى تأخذها الدالة. ما فعلناه هذا يسمى Overload, وهو يعنى أن عدة دوال لهم نفس الأسم لكن نوع عناصرهم مختلفة. فى الكود السابق قمنا بعمل ثلاثة مصفوفات (array) من ثلاثة انواع مختلفة int, double, char. ثم استدعينا كل دالة مع اعطائها النوع المناسب لها. كل هذا طبيعى و معتاد.

لكن الجديد فى C# 2.0 هو اننا نستطيع عمل نفس البرنامج السابق **وذلك بدالة واحدة فقط** , دون الحاجة الى عمل العديد من الدوال.

لو قمنا بتغير اسم الأنواع كلها الى حرف الـ (T) و اعدنا كتابة الدالة بحيث تأخذ النوع T فيصبح الكود كالتالى :

```
Static void PrintArray( T[] inputArray)
```

```
{
      foreach ( T element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}
```

الكود السابق ليس صحيحاً من الناحية اللغوية (الخاصة بلغة البرمجة), لكنه يشرح المعنى ليس أكثر.

اما الكود الصحيح فهو يكتب بالطريقة التالية:

```
Static void PrintArray<T>( T[] inputArray)
{
      foreach ( T element in inputArray )
            Console.Write(element + " ");

      Console.WriteLine("\n");
}
```

لاحظ إضافة <T> بعد أسم الدالة مباشرة, ثم استخدام حرف (T) للتعبير عن النوع بداخل الدالة. طبعاً يمكننا كتابة أى رمز أخرى غير الـ T, لكنك سوف تعتاد على رؤية هذا الحرف كثيراً فى العديد من الكتب التقنية, ووسائل المساعدة المختلفة. لذلك ينصح باستخدام هذا الحرف.

ثم بعد ذلك يمكنك استدعاء تلك الدالة بعدة انواع مختلفة من العناصر:

```
PrintArray( intArray );
PrintArray( doubleArray );
printArray( charArray );
```

فى هذه الحالة يقوم المترجم (Compiler) باستنتاج نوع البيانات التى اعطيت للدالة, ثم تنفيذ الدالة بحيث تتناسب مع النوع المُدخل اليها.

هذا هو المقصود ب Generic. إنشاء كائن للإستخدامات العامة.

Delegates

فلنفترض ان هناك دالة فى كود يتم كتابته, تلك الدالة سوف تستدعى فى مكان محدد فى الكود. والدوال تستدعى بأن يكتب اسمها فى المكان الذى نريده. ثم بعد ذلك نقوم بتشغيل الكود أو ترجمته من قبل المترجم, وكل مرة يتم تشغيل فيها الكود يتم استدعاء الدالة عند نفس النقطة أو المكان الذى كتبنا اسمها فيه. لكن ماذا لو حدث و كان عندنا عدة دوال و نريد أن نستدعى دالة واحدة فقط وفقاً لحدث معين. هناك العديد من الطرق, فمن الممكن ان نستخدم جملة IF, أو جملة Switch. لكن هناك حل أفضل و هو استخدام Delegate.

يعتبر الـ Delegate هو الطريق الذى تعمل به الأحداث. الـ Delegate يعتبر مؤشر للدوال (Pointer). فهو يقوم بتمرير المرجع الخاص بالدول (مكان الدالة فى الذاكرة) و تشغيل تلك الدوال بدون استدعاءها صراحة.

النحو الخاص بالتصريح عن الـ Delegate هو:

```
[access-modifier] delegate result-type identifier( [parameters] );
```

حيث :

access-modifier : معرف الدخول الى الكائن (لمزيد من المعلومات ارجع الى اساسيات لغة C#)

Delegate : كلمة مفتاحية ثابتة

Result-type: نوع البيانات التى ينتجها الـ Delegate و التى تتطابق مع الدالة التى يشير اليها

identifier: أسم الـ Delegate

Paramters: العناصر التى تأخذها الدالة التى سوف تستدعى

المثال التالى تصريح عن Delegates

```
public delegate int myDelegate(double D);
```

فى المثال السابق, قمنا بالتصريح عن Delegate اسمه myDelgegate و الذى يمكن أن نستخدمه لتشغيل اى دالة تعود بقيمة من النوع int و تأخذ عنصر واحد من النوع Double. ولإستخدام هذا الـ Delegate, لابد من عمل نسخة جديدة منه تحتوى على اسم الدالة التى نريد تشغيلها. كالتالى:

```
public int ReturnInt( double D)
{
      // جسم الدالة
}

public boid amethod()
{
      myDelegate aDelegate = new myDelegate(ReturnInt);
}
```

فى بداية الكود قمنا بإنشاء دالة اسمها ReturnInt ونوعها int وتأخذ العناصر D و هو من النوع Duble. ثم فى دالة أخرى قمنا بمعل نسخة من myDelegate و اعطينها الأسم aDelegate ثم حددنا لـ aDelegate الدالة ReturnInt كى يقوم بتشغيلها.

بعد كل هذا, باقى لنا فقط ان نقوم بتمرير قيمة للـDelegate كى يقوم بتشغيل الدالة بها:

```
aDelegate(12345);
```

الدوال المجهولة (Anonymous Methods)

تأمل الكود التالي:

```
private int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] EvenNumber = Array.FindAll(numbers, IsEven);

private bool IsEven(int integer)
{
      Return (integer % 2 == 0);
}
```

فى الكود السابق قمنا بتعريف array اسمها numbers و مكونة من عشرة عناصر من النوع int. و ايضاً قمنا بإنشاء دالة من النوع bool (تعطى false أو true) اسمها IsEven وتأخذ عنصر واحد من النوع int ثم تقوم باختبار هذا العنصر إذا كان زوجى ام فردى, إذا كان زوجى تقوم الدالة بارجاع القيمة true, وإذا كان فردى تعود بالقيمة false. ثم جاء هذا السطر :

```
int[] EvenNumber = Array.FindAll(numbers, IsEven);
```

فى هذا السطر قمنا باستدعاء دالة اسمها FindAll و هى موجودة فى الفئة Array, تلك الدالة تأخذ عنصرين, الأول عبارة عن array و الثانى يجب أن يكون شرط (Condition), و الشرط كما هو معروف فى لغات البرمجة يتم التحقق منه وإذا تحقق الشرط فى تلك الحالة يكون True و إذا لم يتحقق فهو يكون False. لذلك قمنا بإنشاء الدالة IsEven كى تتحقق من الشرط إذا كان False أم True. وقمنا بإستدعائها كعنصر من عناصر الدالة FindAll.

سوف تقوم الدالة FindAll بقرائة المصفوفة التى اعطيت لها, و عند كل عنصر سوف تقوم بتطبيق الدالة IsEven عليه, إذا كانت النتيجة True فسوف تعود الدالة FindAll بذلك العنصر, وإذا كانت النتيجة False لن تعود به. و هكذا حتى تنتهى عناصر المصفوفة كلها. فى حالتنا تلك سوف تعود الدالة FindAll بتلك النتيجة:

```
2, 4, 6, 8, 10
```

لا يهمنا فى كل الكلام السابق سوى الطريقة التى كتبنا بها الكود. فقد قمنا بعمل دالة ثم استدعيناها فى مكان أخر. انظر الى الكود التالى, هو نفس الكود السابق مع بعض التغيرات:

```
private int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] EvenNumber = Array.FindAll(numbers, delegate(int integer
```

```
                    {
                          Return (integer % 2 == 0);
                    }
);
```

هذا الكود يبدو أبسط من الكود السابق, و ربما يكون أكثر فهماً للبعض. فى هذا الكود لم نستدعى الدالة IsEven بل إننا حتى لم ننشأها. الكود الذى كنا قد كتبناه سابقاً بداخل الدالة IseEven قمنا بكتابته مباشرة كعنصر من عناصر الدالة FindAll, مع تغير بسيط فى طريقة الكتابة. فلننظر اليه عن قرب:

```
delegate(int integer
      {
            Return (integer % 2 == 0);
      }
);
```

لقد قمنا بكتابة كلمة Delegate ثم كتبنا الدالة مباشرةً كأنها عنصر بداخل توقيع الـ Delegate. قد يبدو الأمرغير مفهوم للبعض. لكن لو تعاملت من قبل مع الـ Delegates فسوف يكون الأمر فى غاية السهولة لديك. أما وإن لم تكن, فيجب عليك معرفة بعض الشىء عن هذا الـDelegate.

مافعلنا من تغيرات فى الكود هو ما يسمى بـ Anonymous Methods, او الدوال المجهولة. وذلك لأننا أخرجنا جسم الدالة من الشكل الطبيعى للدوال, ثم وضعناه فى مكان استدعاء الدالة مباشرة. لذلك يجب أن ننتبه الى أن تلك الدوال المجهولة لا يمكننا استدعائها فى اى مكان أخر. فلقد أصبحت كأنها قطعة من الكود ليس لها اسم أو توقيع.

yield

جملة yield تأخذ شكلين, إما yield return *expression* أو yield break. لشرح هاذين الشكلين يجب أن نأخذ بعض الأمثلة.

إذا كان لدينا فئة تحتوى على مجموعة من العناصر, وإذا كنا نريد الدخول الى تلك العناصر. فإن هناك العديد من الطرق التى تؤدى الى ذلك. لكن أبسط طريقة هى عمل Iterator (الـ Iterator كائن يقوم بالدخول الى اى مجموعة من العناصر).

انظر المثال التالى :

```
public class DayesOfWeek : IEnumerable
{
      String[] m_Days = { "Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat" };
      public IEnumerator GetEnumerator()
      {
            yield return m_Days[0];
            yield return m_Days[2];
            yield return m_Days[4];
            yield break;
      }
}
```

```
Class TestDaysOfWeek
{
      Static void Main()
      {
            DayesOfWeek week = new DayesOfWeek();
            Foreach ( string day in week )
            {
                  Console.Write(day+ " ");
            }

            IEnumerator myEnumerator = week.GetEnumerator();
            While ( myEnumerator.MoveNext() == true )
            {
                  Console.Write(myEnumerator.Current + " " );
            }
      }
}
```

فى هذا الكود قمنا بإنشاء فئة تسمى DaysOfWeek والتى تقوم بتطبيق الـ interface التى تدعى IEnumerable. (يرجى العودة الى أى مرجع لفهم الـ Interfaces). بتطبيق تلك الـ interface على تلك الفئة, أصبح لدى الفئة دالة تسمى GetEnumerator(). إذا كان بداخل تلك الدالة كلمة yield فإن المترجم يحدد تلك الدالة على إنها iterator ويقوم بتوليد فئة جديدة تقوم بتطبيق الـ interface التى تدعى IEnumerator وايضاً يقوم بتوليد الدوال التى تحتويها تلك الـ interface و هما MoveNext و Dispose. ثم بعد ذلك قمنا بعمل Loop باستخدام الـ iterator. فى كل مرة يتم فيها تنفيذ دالة MoveNext الموجودة بداخل الـ iterator يتم تنفيذ جملة yield واحدة ثم يتوقف. ثم فى الدورة الثانية يتم تنفيذ الجملة الثانية ثم يتوقف وهكذا حتى يصل الى جملة yield break وعندها يتوقف نهائياً.

دعنا نشرح الكود شرحاً أكثر تفصيلاً:

```
IEnumerator myEnumerator = week.GetEnumerator();
```

هنا قمنا باشتقاق الكائن myEnumerator من الفئة التى يقوم المترجم بتوليدها.

myEnumerator.MoveNext()

هنا يتم ايجاد أول جملة yield. إذا وجد المترجم جملة yield break أو انتهى المجال الذى يدور فيه فسوف تعود دالة MoveNext بالقيمة false وتنتهى جملة while.

myEnumerator.Current

تلك الجملة تحتفظ بالقيمة التى تعود بها جملة yield.

أما نتيجة تنفيذ هذا البرنامج فهى كالتالى:

```
Sun Tue Thr
```

ميزات C# 3.0

الإصدار C# 3.0 حرك لغة C# فى إتجاه اللغة الوظيفية. وذلك بتقديم أسلوب أكثر تصريحاً. و تستخدم LINQ جميع الميزات الجديدة فى C# 3.0 تقريباً, و تلك المميزات تجعلنا قادرين على كتابة كود أكثر بساطة ووضوح.

استنتاج النوع (Local Type Inference)

إستنتاج النوع يعتبر ميزة رائعة فى أى لغة. فهو يجعلك تكتب الكود بمنتهى الراحة و دون تحديد نوع المتغير او الكائن الذى تتعامل مع. فالمترجم الخاص باللغة يقوم باستنتاج نوع المتغير أو الكائن, وذلك بتحليل القيمة الموجودة فى المتغير أو الكائن و تحديد نوعها.

تقدم C# 3.0 ميزة إستنتاج النوع عن طريق كتابة كلمة var بدلاً من كتابة نوع البيان. انظر المثال التالى:

```
int a = 5;
var b = a;
```

قمنا بتعريف متغير من النوع int و اعطيناه القيمة 5, ثم عرفنا متغير مجهول النوع و اعطيناه قيمة المتغير a. تلقائياً يقوم المترجم بتحليل البيانات الموجودة فى المتغير a ثم ظبط نوع المتغير B كى يتوافق معها, وبالتالى بعد تلك العملية يتحول المتغير B الى النوع int. وذلك عن طريق الإستنتاج. انظر الكود التالى:

```
int a = 5;
int b = a;
```

هذا الكود مكافىء للكود السابق تماماً, لكن الفرق هنا إننا حددنا نوع المتغير B مسبقاً, لذلك لن يستنتج المترجم شيئاً فالنوع محدد امامه.

بالنسبة لبعض الناس. تعتبر ميزة استنتاج الكود اداة للمبرمجين الكسالى. ومع ذلك, استنتاج الكود يعتبر الطريقة الوحيدة التى يمكنك بها تعريف متغيرات مجهولة النوع, كما سوف نرى لاحقاً.

كلمة var يمكن أن تستخدم بداخل المجال (Scope) الحالى فقط. لتوضيح ذلك انظر الكود التالى:

```
public void ValidUse( Decimal d) {
      var x = 2.3             // double
      var y = x               // double
      var r = x / y           // double
      var s = "Sample"        // string
      var I = s.Length;       // int
      var w = d;              // decimal
      var p = default(string) // string
}
```

فى الكود السابق قمنا بتعريف متغيرات var داخل جسم الدالة *ValidUse* دون أى مشكلة, الكود التالى يوضح الماكن التى لا يسمح فيها بتعريف متغيرات var.

```
class VarDemo {

      var k =0;          // لا يجوز تعريف متغير مجهول فى جسم الفئة مباشرة

      // لا يجوز للعناصر الموجودة فى توقيع الدالة أن تكون مجهولة
      public void InvalidUseParameter( var x) {}

      // لا يجوز تعريف دالة مجهولة النوع
      public var InvalidUseResult() {
            return 2;
      }

      public void InvalidUseLocal() {
            var x;            // = خطأ فى النحو لا يوجد علامة
            var y = null      //  null لا يمكن استنتاج نوع المتغير من القيمة
      }
}
```

فى الكود السابق, المفترض إن المترجم يستطيع استنتاج نوع المتغير K وذلك من خلال القيمة المبدئية للمتغير. لكن هذا غير مسموح به فى هذا المكان من الكود. ايضاً الدالة *InvalidUseResult* من الممكن استنتاج نوع النتيجة التى تعود بها, لكن هذا غير مسموح به, فلابد من تحديد نوع الدلة من قبل المبرمج.

تعبيرات لمدا (Lambda Expressions)

كما ذكرنا سابقاً إن قطعة الكود عبارة عن تحزيم لمجموعة من الجمل البرمجية بهذف تنفيذها مع بعض فى وقت واحد او عدم تنفيذها على الإطلاق. لذلك عندما نحتاج الى كتابة قطعة من الكود بداخل البرنامج فنقوم بعمل دالة تحتوى على قطعة الكود.

الكود التالى عبارة برنامج بسيط, قمنا بإنشاء List تحتوى على عدة اسماء. ثم أنشأنا دالة للبحث عن أسم معين فى الـ List.

```
class Program
{
    static void Main(string[] args)
    {
        List<string> names = new List<string>();
        names.Add("Dave");
        names.Add("John");
        names.Add("Abe");
        names.Add("Barney");
        names.Add("Chuck");
        string abe = names.Find(IsAbe);
```

```
        Console.WriteLine(abe);
    }
    public static bool IsAbe(string name)
    {
        return name.Equals("Abe");
    }
}
```

الكود السابق عبارة عن دالتين, الدالة الثانية اسمها IsAbe تحتوى على سطر واحد من الكود – يطلق عليه ايضاً قطعة كود – وتتعامل مع عنصر يدعى name و نوعه string.

فى الإصدار الثانى من لغة C# تم تقديم الدوال المجهولة (Anonymous Methods) و التى تحدثنا عنها سابقاً. و هى مفيدة فى حالة إذا كانت قطعة الكود لن تستخدم إلا مرة واحدة, لذلك يمكننا التعديل على الكود السابق ليصبح كالتالى:

```
class Program
{
    static void Main(string[] args)
    {
        List<string> names = new List<string>();
        names.Add("Dave");
        names.Add("John");
        names.Add("Abe");
        names.Add("Barney");
        names.Add("Chuck");
        string abe = names.Find(delegate(string name)
                        {
                            return name.Equals("Abe");
                        });
        Console.WriteLine(abe);
    }
}
```

لقد قمنا بالغاء الدالة IsAbe نهائياً وكتبنا الكود الذى كان بداخلها فى المكان الذى استدعيناها فيه فى الكود الأول. أى اننا بدلاً من عمل دالة فى مكان ثم استدعائها فى مكان أخر, قمنا بكتابة الكود مباشرة دون عمل تلك الدالة.

اما Lambda Expressions فتقوم بتسهيل العملية أكثر من ذلك:

```
class Program
{
    static void Main(string[] args)
    {
        List<string> names = new List<string>();
        names.Add("Dave");
        names.Add("John");
        names.Add("Abe");
        names.Add("Barney");
        names.Add("Chuck");
        string abe = names.Find((string name)=> name.Equals("Abe"));
        Console.WriteLine(abe);
    }
```

```
}
```

فى الكود السابق, استغنينا عن الـ Delegate وقمنا بكتابة تعبير لمدا مباشرةً :

```
(string name)=> name.Equals("Abe")
```

ولأن Lambda Expression ذكية بما فيه الكفاية لإستنتاج نوع العنصر, لذلك يمكننا الإستغناء عن تعريف نوع العنصر فى بداية التعبير:

```
name => name.Equals("Abe")
```

ويمكنك ان تستبدل name باسم اقل من هذا, فهذا التعبير لن يستخدم إلا مرة واحدة, اى انك لن تحتاج الى استخدام هذا العنصر مرة أخرى. لذلك يمكن استخدام حرف واحد فقط فى Lambda Expression:

```
n => n.Equals("Abe")
```

تعبيرات لامدا تكتب على هذا الشكل دائماً

```
X => f(X)
```

حيث X يمثل العنصر, و f(X) تمثل الدالة أو المعادلة أو التعبير الذى يستخدم العنصر X.

الدوال الإضافية (Extension Methods)

كما هو واضح من الأسم, فإن الدوال الإضافية تعمل كامتداد للكائنات. اى انه إذا كان عندنا فئة تحتوى على دالتين, و أستنسخنا من تلك الفئة كائن. يمكننا إضافة دوال لهذا الكائن عن طريق Extension Methods.

المثال التالى يقوم بإضافة دالة جديدة تدعى IsVaildEmailAddress لنسخة من الفئة String:

```
class Program
{
    static void Main(string[] args)
    {
        string customerEmailAddress = "test@test.com";
        if (customerEmailAddress.IsValidEmailAddress())
        {
            // Do Something...
        }
    }
}
public static class Extensions
{
    Public static bool
    IsValidEmailAddress(this string s)
    {
        Regex regex = new
        Regex(@"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
        return regex.IsMatch(s);
    }
}
```

عندما نأخذ نسخة من الفئة String فهذا يعنى إننا نعرف متغير من النوع string كما هو واضح فى الكود السابق, فهذا السطر يقوم بتعريف string يدعى customerEmailAddress.

```
string customerEmailAddress = "test@test.com";
```

والأن نريد إضافة دالة للفئة String تلك الدالة تقوم بالتحقق من إذا كان هذا الإيميل مكتوب بطريقة صحيحة ام لا. فقمنا بكتابة الدالة التالية:

```
IsValidEmailAddress(this string s)
{
    Regex regex = new
    Regex(@"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
    return regex.IsMatch(s);
}
```

لا يهم ما الذى كتب فى جسم الدالة, الذى يهمنا الأن هو العناصر التى توجد فى توقيع الدالة. كما نلاحظ أن الدالة تحتوى على عنصر واحد و هو s و هو من النوع String. وما نلاحظة ايضاً أن هناك كلمة this تسبق كلمة string. بوجد كلمة this قبل العنصر الأول فى اى دالة, هذا يعنى إن تلك دالة إضافية (Extension Method), و سوف تضاف لـ String. ويجب أن تكون الدالة الإضافية Static و كذلك الكائن الذى سوف تضاف اليه.

Object Initialization Expressions

افترض أن لدينا الفئة التالية:

```
public class Customer
{
    private int _id;
    public int Id
    {
        get { return _id; }
        set { _id = value; }
    }

    private string _name;
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    private string _city;
    public string City
    {
        get { return _city; }
        set { _city = value; }
    }

    public Customer() {}

}
```

إذا اردنا إشتقاق كائن من تلك الفئة فنحن نقوم بكتابة الشطر التالى :

```
Customer c1 = new Customer();
```

فى تلك الحالة اصبح لدينا كائن يسمى C1 و هو مشتق من الفئة Customer. الخطوة التالية هى إعطاء قيم لخصائص هذا الكائن. و تلك الخصائص موضحة فى الكود الخاص بالفئة بأعلى. و إعطاء القيم يتم كالتالى:

```
c1.ID = 1;
c1.Name = "Ahmed";
c1.City = "Cairo";
```

تلك هى العملية الكاملة لإشتقاق كائن و تخصيص قيم لخصائصة. لكن باستخدام ميزة Object Initialization يمكننا تحديد قيم لخصائص الكائن أثناء اشتقاقه مباشرة, كالتالى:

```
Customer c1 = new Customer { Id = 1, Name = "Ahmed",  City="Cairo" };
```

و يمكننا عمل مجموعة كائنات مرة واحدة و تخزين تلك الكائنات فى قائمة (List), كالتالى:

```
List<Customer> listofCustomers =
new List<Customer> {
            { Id = 1, Name = " Ahmed", City = "Cairo" },
            { Id = 2, Name = " Mohamed", City = "Alex" },
            { Id = 3, Name = "Mahmoud", City = "Tanta" },
      };
```

الأنواع المجهولة (Anonymous Type)

انظر الى الكود التالى:

```
var p1 = new { Name = "DVD", Price = 3 };
```

فى هذا الكود قمنا بإنشاء فئة تحتوى على خاصيتين Name و Price, واعطينا كل خاصية قيمة.

هذا الكلام كان يعتبر ضرباً من الخرافات سابقاً. لكنه الأن حقيقى مائة بالمائة. فهذا الكود بالفعل هو كود إنشاء فئة. لكنها بدون اسم و بدون الهيكل المعتاد لإنشاء الفئات. هذا هو الجديد فى الموضوع. فالمترجم يقوم بتوليد اسم لهذه الفئة ويقوم بعمل الهيكل المعتاد بدلاً منك. ولا داعى أن أقول ان هذه الفئة لا يمكن استنساخها. فهى تستخدم فى مكانها فقط.

Query Expression

هل كتبت استعلام SQL من قبل؟. هل كان يبدو مثل هذا الإستعلام :

```
var query =
    from c in customers
    where c.Discount > 3
    orderby c.Discount
    select new { c.Name, Perc = c.Discount / 100 };
```

أعتقد انه مشابه لهذا الإستعلام الذى كتبته بـ SQL. لكنه مختلف. فهذا استعلام بلغة C#. شرح هذا الإستعلام يأتى فى الفصل التالى. فهذا الإستعلام من اختصاص LINQ, و الفصل التالى نشرح فيه LINQ.

2

الفصل الثانى

أساسيات LINQ

نتعرف فى هذا الفصل على كيفية كتابة تعبير استعلامى باستخدام LINQ.

ما هى LINQ ؟

باختصار شديد و ببساطة أشد, تعتبر LINQ تقنية حديثة تهدف الى توحيد طرق الأستعلام عن البيانات من مصادرها المختلفة... يبدوا ان التعريف لم يكن بسيطاً كما كنا نتوقع... فلنبسط الموضوع قليلاً.

ما هى تلك المصادر المختلفة للبيانات التى يتحدث عنها التعريف. أعتقد انك تعلم ان هناك ما يسمى بقاعدة البيانات, وهى عبارة عن ملف تخزن فيه البيانات بشكل معين بهدف استرجاع تلك البيانات و ادارتها وهكذا. وتعلم ايضاً ان هناك ملفات تسمى Spreadsheets, هذه الملفات تقوم بتخزين البيانات بشكل معين بحيث يسهل اجراء عليها العمليات الحسابية أو الإحصائية أو اياً ما كان. و هناك ايضاً ملف نص txt, والذى نخزن فيه البيانات بشكل معين بهدف قرائتها فيما بعد أو ارسالها الى اى جهة أو اى سبب اخر. و هناك العديد من أشكال تخزين البيانات الموجودة فى عصرنا الحالى... لكن! .... هل تعلم انه لكى تحصل على البيانات الموجودة فى اى مصدر من المصادر السابقة فأنه يلزم ان تستخدم طريقة مختلفة لكل مصدر!. طبعاً انت تعلم هذا جيداً. فلكى نحصل على بيانات مخزنة فى ملف spreadsheet فنحن نستخدم برنامج مثل Excel أو math أو اى برنامج يدعم ملفات spreadsheets. ولكى تحصل على بيانات من قاعدة بيانات فأنت تستخدم برنامج ادارة قواعد بيانات مثل SQL Server أو Oracle. وهكذا تجد ان كل مصدر بيانات له طريقة لإستخراج البيانات منه.

هنا تظهر تقنية LINQ او Language integrated Query, وتقدم حلاً ممتاز للوصول الى البيانات من اى مصدر اياً كان.. هل تعلم ما هى الـArray؟. طبعاً تعلم فهى كائن يتم توليده فى الذاكرة و يتم تخزين فيه مجموعة من البيانات من نفس النوع. هل تعلم إنه يمكنك الإستعلام عن البيانات الموجودة فى الـ array باستخدام LINQ. ايضاً يمكن الإستعلام عن بيانات فى الكود الذى تكتبه. حيث يمكنك الإستعلام عن دوال معينه قمت انت بكتابتها. والكثير و الكثير.... ربما تستطيع LINQ يوماً ما أن تستعلم عن البيانات الموجودة فى المخ البشرى ☺.

أنواع LINQ

أما عن أنواع LINQ فهى LINQ to SQL – LINQ to XML – LINQ to Objects.

بعد كل تلك المقدمات السابقة عن C# و عن LINQ, ولم ندخل فى صلب الموضوع بعد... أعتقد انه الوقت المناسب كى نتحدث قليلاً عن كيفية استخدام LINQ و العمل بها. لكن!. اياك أن تكمل إن لم تفهم ما سبق وإلا لن تفهم شيئاً مما سيأتى. أو يمكنك أن تجرب بنفسك, فاعتقد انك لست من الذين يأخذون بالنصيحة☺.

استعلامات LINQ

Query Syntax

كى نفهم النحو الخاص بالإستعلام (Query Syntax), يجب أن نبدأ بمثال بسيط. الكود التالى عبارة عن فئة اسمها Developer و تحتوى على ثلاث متغيرات Name, Language و Age:

```
public class Developer
{
	public string Name;
	public string Language;
	public int Age;
}
```

تخيل أنك هناك array من هذه الفئة بداخل الذاكرة. اى ان عناصر الـ array عبارة عن مجموعة من هذا الكائن Developer. و تريد أن تستعلم عن تلك الكائنات الموجودة بتلك الـarray. انظر الى الكود التالى:

```
using System;
using System.Linq;
using System.Collections.Generic;

class app
{
	start void Main()
	{
		Developer[] developers = new Developer[]
		{ new Developer { Name = " Ahmed", Language="C#" },
		new Developer { Name="Mohamed", Language="C#",
		new Developer { Name="Taha", Languag="VB.NET"} };

		IEnumerable<string> developersUsingCsharp =
			from d in developers
			where d.Language =="C#"
			select d.Name;

		froeach ( string s in developersUsignCsharp )
		{
			Console.WriteLine(s);
		}
	}
}
```

هل لاحظت وجود Object Initialization؟.. لا عليك مجرد سؤال. ما يهمنا فى هذا الكود هو الأسطر المكتوبة بخط سميك (Bold) . نعم تلك الأسطر:

```
from d in developers
where d.Language =="C#"
select d.Name;
```

هذا هو الإستعلام. من أول نظرة يبدو وكأنة استعلام SQL, لكن مع قليل من التدقيق تجد انه مختلف بعض الشىء.

قبل أن نكمل يجب أن تعلم ما هو Query Expression. إن الإستعلام السابق جزء من query Expression. هو ليس ناقص بل هو جزء منه. فـ Query Expression يتكون من أجزاء عديدة يمكننا أخذ بعضها و ترك البعض الأخر. تلك الأجزاء تسمى Operators. وكل Operator يقوم بوظيفة محددة. فكما نرى فى الإستعلام السابق الجزء الخاص بـ Select:

```
select d.Name;
```

هذا الجزء خاص بالـ Operator المسمى Select. و الذى يقوم باحضار البيانات من مصدر البيانات. وهناك ايضاً الجزء الخاص بالـ from Operator:

```
from d in developers
```

حيث يقوم هذا الـ Operator بتحديد الكائن الذى سوف نأخذ منه البيانات. ويجب أن علم ان الكائن الذى سنأخذ منه البيانات يجب أن يقوم بتطبيق IEnumerable<t> interface. اما هذا الجزء من الإستعلام:

```
where d.Language =="C#"
```

فهو الـ Operator الذى يحدد الشروط الواجب توافرها فى البيانات التى سوف نستعلم عنها. اى انه يقوم بعملية فلترة للبيانات الموجودة وايجاد البيانات ذات المواصفات المحددة له.

الإستعلام السابق يمكننا كتابتة بطريقة أخرى تسمى Expression Tree:

```
IEnumerable<string> expr =
      developers
      .Where (d => d.Language == "C#" )
      .Select (d => d.Name);
```

لاحظ أن Where و Select مكتوب بعدهم Lambda Expression. هذه الـ Lambda Expression تترجم الى مجموعة من generic delegate.

نحو الإستعلام الكامل (Full Query Syntax)

كما تعلمون دائماً توجد مشاكل عند ترجمة المصطلحات العلمية, فالعلم يتحدث الإنجليزية لذلك فالمصطلحات فى الأساس باللغة الإنجليزية. وعند محاولة ترجمة تلك المصطلحات تنتج أسماء غريبة واحياناً تسبب ارتباك لمتلقى المعلومة. لذلك, نصيحة... اهتم بالمصطلح الإنجليزى فالمصطلح العربى غالباً ما يكون غير مناسب. انا شخصياً لا استسيغ ترجمة Full Query Syntax. فهى باللغة العربية تكاد تكون غير مفهومة. المهم... دعك من كل هذا و تعالى نرى ما هو هذا الـFull Query Syntax.

اعتقد انك لاحظت اننا نكتب كلمة From فى اول الإستعلام. واعتقد انك تعلم إن إستعلام SQL نتكب فى اوله Select. لماذا جعلت مايكروسوفت كلمة From فى اول الإستعلام وجعلت كلمة Select فى أخر الإستعلام. هذا موضوع يطول شرحه. وبما اننا نملك الكثير من الوقت فلنتحدث عنه قليلاً.

هل سمعت يوماً ما عن تقنية تسمى IntelliSense ؟. واضح انك لم تسمع... حسناً ... هل استخدمت برنامج Visual Studio من قبل ؟. واضح انك استخدمته. جميل... عند كتابة الكود فى هذا البرنامج, هلا لحظت انه فى بعض الأحيان يقوم البرنامج بإظهار قائمة بالعناصر المحتملة التى تستطيع كتابتها فى تلك المنطقة من الكود. مثلاً. عند كتابة هذا السطر:

```
Using System.Xml;
```

عند كتابتك لكلمة System ووضع النقطة التى بعدها تجد البرنامج يظهر لك قائمة تحتوى على كل العناصر المتفرعة من System التى تستطيع كتابتها بعد كلمة System. ما الذى أخبر Visual Studio بتلك العناصر المحتملة, ولماذا هى تختلف تبعاً للعنصر الذى نكتبه. هذا هو مايسمى IntelliSense. هل وضحت الصورة الأن. لنعود الى الإستعلام ولماذا From قبل Select.

ما الذى نكتبة بعد كلمة From ؟. اثناء استخدامنا لـ SQL تعودنا ان نكتب اسم الجدول الذى نريد احضار البيانات منه بعد كلمة From. وفى LINQ نفس الأمر بالضبط, نحن نكتب اسم الجدول أو الكائن الذى يحتوى على البيانات. وبعد كلمة Select نقوم بتحديد العناصر التى سوف تسترجع من هذا الجدول أو الكائن. لو كتبنا كلمة Select فى أول الإستعلام بالنسبة لـ LINQ فلن يستطيع Visual Studio معرفة اسماء العناصر المحتملة لأنه لا يعرف من اى مصدر بالضبط سوف تأتى البيانات. اما لو كتبنا كلمة From فى اول الإستعلام فبهذا نخبر Visual Studio مصدر البيانات أولاً. بالتالى يتعرف Visual Studio على مصدر البيانات و يعرف ما هى العناصر المحتملة التى يمكن استرجاعها وبالتالى يقدم لنا قائمة بتلك العناصر و نختار منها ما نريد.... اعتقد انك فهمت الأن لماذا From قبل Select.. عامة إذا كان الموضوع صعب الفهم فلا عليك. اكتب From قبل Select واعتبره أمر عسكرى.

نعود الى الموضوع الأصلى وهو Full query syntax. الإستعلام الكامل يكون على هذا الشكل:

from *id* **in** *source*

{ from *id* **in** *source* /

Join *id* **in** *source* **on** *expr* **equals** *expr* [**into** *id*] |

Let *id* = *expr* |

Where *condition* |

Orderby *ordering*, *ordering*, ... [**Ascending** | **Descending**] **}**

Select *expr* |

Group *expr* **by** *key*

[**into** *id query*]

الشرح

from *id* **in** *source*

حيث:

id: اسم العنصر. حيث نقوم بتحديد اسم للعنصر الموجود فى المصدر

source: المصدر

مثال للشرح:

```
from p in Products
```

فى السطر السابق قمنا بتحديد اسم للعنصر اسمه p وهذا العنصر يأتى من جدول Products.

ملحوظة بسيطة. يمكن لعبارة form ان تأتى بعدها عبارة form اخرى. بل انه يمكن ان يأتى بعدها عدة عبارات form وليس عبارة واحدة. ويمكن ايضاً ان يأتى بعدها عدة عبارات join.

Select *expr*

حيث expr هو الحقل أو الحقول التى سوف نسترجعها من الجدول, و هو يكتب فى شكل expression.

Group *expr* **by** *key*

حيث key هو الحقل الذى سوف نعمل به group.

Let *id = expr*

تستخدم لعمل استعلامات فرعية أو كما يطلق عليها Sub-Queries.

Where *condition*

حيث condition هو الشرط أو مجموعة الشروط. حيث تقوم عبارة where بعمل فلتر للبيانات وذلك بالبحث عن بيانات لها شروط محددة.

Join *id* **in** *source* **on** *expr* **equals** *expr*

وهى تقوم بتحديد شكل العلاقة بين الكينونات أو الجداول.

Orderby *ordering*, *ordering*, ...[**Ascending | Descending**]

وهى تقوم بترتيب عرض البيانات إما تنازلياً أو تصاعدياً. على حسب قيمة حقل أو اكثر من حقل.

into *id query*

وضع النتيجة فى مكان مؤقت للإستعلام منها. شىء ما مشابه لعملية الإستعلامات الفرعية.

ما سبق عبارة عن مرجع سوف نرجع اليه عند الحاجة الى كتابة استعلام. وسوف نفهم كل شىء عن تلك العبارات اثناء الشرح.... لا تقلق فنحن لم نبدأ شرح بعد .

Query Operators

هذه المرة لن أقوم بترجمة معنى Operators, لأن ترجمتها الحرفية سوف تسبب بعض الإرتباك فكلمة Operator تعنى المشغل او العامل. لكننا نريدها كما هى Operator.

يمكننا شرح المقصود من Operators عوضاً عن ترجمتها. فهى عبارة عن دوال تأتى جاهزة مع LINQ و تلك الدوال يتم تخصيصها الى اى استعلام تقوم بإنشائه. اى إن الإستعلام يصبح قادر على استخدم تلك الدوال فى عملياته الخاصة.

الجدول التالى يوضح جميع الـ Operators التى تأتى مع LINQ.

جدول 2.1: LINQ Query Operators

Aggregate	All	Any	Average
Cast	Concat	Contains	Count
DefualtIfEmpty	Distinct	ElementAt	ElementAtOrDefault
Empty	EquallAll	Except	First
FirstOrDefault	Fold	GroupBy	GroupJoin
Intersect	Join	Last	LastOrDefault
LongCount	Max	Min	OfType
OrderBy	OrderByDescending	Range	Repeat
Reverse	Select	SelectMany	Single
SingleOrDefault	Skip	SkipWhile	Sum
Take	TakeWhile	ThenBy	TheyByDescending
ToArray	ToDictionary	ToList	ToLookup
ToSequence	Union	Where	

أراك قد قمت بعد الأعمدة ووجدتهم 4 وعددت الصفوف ووجدتهم 13 ثم قمت بضرب الأعمدة فى الصفوف ووجتهم 52 ثم قمت بطرح واحد من الناتج فأصبح 51... كنت سأخبرك على أى حال انهم 51 Operator. والمفاجأة انى سأشرحهم كلهم. هيا نبدأ.

العناصر (Elements) التى سوف تذكر فى شرح الـ Operators هى عناصر مجموعات (collections) مثل المصفوفات أو الـ List او ما شابه. اى انها عناصر تحتوى على مجموعة عناصر.

Aggregate

هل سمعت عن الجمع المتكرر؟. مثلاً عندك الأرقام الأتية (6 ,5 ,4 ,3 ,2 ,1) و تريد جمعها جمع متكرر. وهو بأن تجمع أول عنصر مع ثانى عنصر و الناتج تجمعه مع ثالث عنصر و الناتج تجمعه مع رابع عنصر وهكذا حتى أخر العناصر. يقوم Aggregate operator بتلك العملية سواء كانت جمع أو ضرب أو اياً ما كانت. انظر المثال التالى:

```
int[] ints = {1, 2, 3, 4, 5, 6};
var query = from ...; // لن أكتب الإستعلام نفسه فلن نحتاجه فى الشرح
int sum = ints.Aggregate( (a,b) => a + b );
int product = ints.Aggregate( 1, (a,b) => a * b );
int sump1 = ints.Aggregate( 0, (a,b) => a + b, r => r + 1 );
```

بعد تنفيذ هذا الكود تصبح قيمة المتغير sum تساوى 21 وتصبح قيمة المتغير product تساوى 720 وتصبح قيمة المتغير sump1 تساوى 22. كان هذا مثال بسيط.

All

هل كل العناصر تحقق شرط معين. هذه هى فائدة All Operator. فلنفترض انك تريد معرفة هل كل الـ Doctors لديهم رقم هاتف. وهل كل الـ Doctors الذين يسترجعهم الإستعلام لديهم رقم هاتف.

```
Doctors doctors = new Doctors();
Var query = from doc in doctors ...;
bool allHavePhone = doctors.All(doc => doc.Phone > 0 );
bool theseHavePhonr = query.All(doc => doc.Phone > 0 );
```

طبعاً النتيجة تكون True أو False.

Any

يقوم هذا الـ operator بفحص مصدر البيانات و البحث عن اى عنصر يحقق بعض الشروط المطلوبة. فلنفترض انك تريد معرفة هل هناك أى Doctor يعيش فى مدينة القاهرة:

```
Doctors doctors = new Doctors();
bool inCairo = doctors.Ant( doc => doc.City == "Cairo" );
```

النتيجة True أو False.

Average

متوسط القيم. مجموع القيم على عددها:

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
decimal?[] values = { 1, null, 2, null, 3, 4 };
Doctors doctors = new Doctors();
var query = from ...;
double avg1 = ints.Average();
decimal? avg2 = values.Average();
double avgYears = doctors.Average(
            doc => DateTime.Now.Subtract(doc.StartDate).Days / 365.25 );
var avg = query.Average();
```

ها هناك ما يحتاج الى شرح ؟... اعتقد ان avgYears تحتاج الى بعض التوضيح. هنا نقوم بمعرفة الوقت الحالى ثم طرح منه تاريخ بدأ العمل (التاريخ الذى بدأ فيه الدكتور العمل) ثم قسمة الناتج على 365.25 وهو عدد أيام السنة ويكون الناتج مثلاً 1.5 هذا معناه ان الدكتور يعمل منذ عام ونصف.

بالنسبة للمتغير avg1 قيمته تساوى 3.5 وهو مجموع قيم ints مقسومة على عددها. والمتغير avg2 تكون قيمته 2.5 وهو مجموع القيم الغير فارغة فى values مقسومة على عددها وهى هنا 4 قيم. اما المتغير avg فلا يعلم قيمته إلا الله, فنحن لم نحدد استعلام محدد كى يحسب لنا متوسطه;).

Cast

يقوم يتحويل أنواع العناصر الى نوع محدد.

```
ArrayList al = new ArrayList();
al.Add("abc");
al.Add("def");
al.Add("ghi");

var strings = al.Cast<string>();
```

الكود السابق مجموعة من العناصر من النوع ArrayList تم تحويلها الى النوع string.

Concat

يقوم بربط (Concatenate) عنصرين ببعض فيصبحا عنصر واحد.

```
int[] ints1 = { 1, 2, 3 };
int[] iints2 = { 4, 5, 6 };

var query1 = from ...;
```

```
var query2 = from ...;


object[] objects1 = { "abc, "dfe" };
object[] objects2 = {1, 2, 3};

var all = ints1.Concat(ints2);
var results = query1.Concat(query2);
var result = objects1.Concat(objects2);
```

بعد تنفيذ هذا الكود تصبح قيمة المتغير all تساوى (1, 2, 3, 4, 5, 6). لأن Concat Operator قام بوصل العنصرين ints1 و ints2 معاً فأصبحا عنصر واحد. والمتغير result لا نعلم قيمته لأننا لم نكتب استعلام. ولا داعى لقول هذا فى كل مرة.

Contains

يبحث بداخل العنصر هل يحتوى على قيمة معينه.فلنفترض انك تريد معرفة هل هناك Doctor اسمه Ahmed مازال يعمل فى المستشفى؟. فى تلك الحالة نقوم بعمل نسخة من الكائن Doctor ونعطيها القيم التى نبحث عنها , ونرى إذا كان العنصر يحتوى عليها ام لا:

```
Doctors doctors = new Doctors();
bool docExists = doctors.Contain( new Doctor("Ahmed", ... ) );
```

هناك طريقة أخرى, هى اننقوم بعمل الإستعلام ثم البحث بداخل نتيجة الإستعلام:

```
var query = from doc in doctors
            select doc.Name
bool docExists = query.Contain("Ahmed");
```

Count

يقوم بعد العناصر التى بداخل اى مجموعة (Collection).

```
int[] ints = {1, 2, 3, 4, 5, 6 };
decimal?[] values = {1, null, 2, null, 3 };
IEnumerable<Doctor> doctors = new Doctors();
Var query = from ...;

int count1 = ints.Count();
int count2 = values.Count();
int count3 = doctors.Count;
int count4 = doctors.Count( doc => doc.City == "Cairo" );
int count = query.Count();
```

نتائج تنفيذ الكود هى:

count1 = 6, count2 = 5, count3 = 12, count4 = 5

DefualtIfEmpty

إذا كان هناك عنصر فارغاً فيتم وضع قيمة افتراضية, وإذا لم تحدد تلك القيمة فيتم وضع القيمة الإفتراضية الخاصة بنوع بيانات العنصر:

```
int[] ints1 = {1, 2, 3, 4, 5, 6 };
int[] ints2 = { };
var query = from ...;

var ints = ints1.DefaultIfEmpty();
var Zero = ints2. DefaultIfEmpty();
var minus1 = ints2. DefaultIfEmpty(-1);
var result = query. DefaultIfEmpty();
```

المتغير ints تصبح قيمته نفس قيمة المتغير ints1 وذلك لأنه ليس فارغاً. اما المتغير Zero فتصبح قيمته 0 لأن المتغير ints2 فارغاً وهو من النوع int والقيمة الإفتراضية لـint هى صفر. والعنصر minus1 تصبح قيمته 1- وذلك لأن العنصر ints2 فراغ وقمنا بتحديد قيمة إفتراضية إذا كان فارغاً وهى 1-.

Distinct

يقوم بإرجاع النتائج بدون اى تكرارات:

```
int[] ints = { 1, 2, 2, 3, 2, 3, 4 };
var query = from ...;

var distinctInts = ints.Distinct();
var distinctResult = query.Distinct();
```

بعد تنفيذ الكود تصبح قيمة distinctInts تساوى (1, 2, 3, 4) اى انه تم حذف كل العناصر المتكررة.

ElementAt

يقوم بإرجاع القيمة الموجودة فى رتبة محددة. مثلاً القيمة الثانية أو الثالثة:

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var query = from …;

int third = ints.ElementAt(2);
Doctor doctor = doctors.ElementAt(2);
var result = query.ElementAt(i);
```

من المعروف إن الرتب فى #C تبدأ من القيمة 0 اى انه فى العنصر ints رقم (1) يعتبر فى الرتبة رقم 0 ورقم (2) فى الرتبة رقم واحد وهكذاً. بتنفيذ هذا الكود تصبح قيمة المتغير third تساوى 2.

ElementAtOrDefault

ارجاع القيمة الموجودة فى رتبة محددة مع الأخذ فى الإعتبار انه قد يكون هناك قيم فارغة.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var query = from …;

int x1 = ints1.ElementAtOrDefault(2);
int x2 = ints1.ElementAtOrDefault(6);

int x3 = ints2.ElementAtOrDefault(0);
Doctor doc1 = doctors.ElementAtOrDefault(2);
Doctor doc2 = doctors.ElementAtOrDefault(-1);

var result = query.ElementAt(i);
```

قيمة المتغير x1 تساوى 3. وقيمة المتغير x2 تساوى 0 وذلك لأن العنصر ints لا يوجد به رتبة سادسة فهو يحتوى على ستة عناصر فقط تبدأ من الرتبة رقم صفر وتنتهى بالرتبه رقم خمسة.

Empty

يقوم بإنتاج مجموعة عناصر فارغة:

```
var emptyDocs = System.Query.Sequence.Empty<Doctor>();
foreach( var doc in emptyDocs)
      Console.WriteLine(doc);
```

هذا الـ Operator مفيد إذا ما كان هناك حاجة لمجموعة فارغة من نوع معين.

EqualAll

يقوم هذا الـ Operator بمقارنة مجموعتين من العناصر و يتأكد إذا كانوا يحملان نفس العناصر ونفس العدد ام لا.

```
var query1 = from ...;
var query2 = from ...;
bool equal = query1.EqualAll(query2);
```

Except

يقوم بمقارنة مجموعتين من العناصر ويسترجع القيم الموجودة فى المجموعة الأولى وغير موجودة فى المجموعة الثانية بدون تكرارات.

```
int[] intsS1 = { 1, 2, 2, 3, 4, 5, 6 };
int[] intsS2 = { 1, 3, 6, 7 };
var diffInts = intsS1.Except(intsS2);
```

تصبح قيمة المتغير diffInts تساوى (2, 4, 5).

First

يقوم باسترجاع أول قيمة فى مجموعة عناصر, بشرط أن تكون تلك المجموعة غير فارغة.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var query = from ...;

int first = ints.First();
Doctor doctor = doctors.First( doc => doc.City == "Cairo" );
var result = query.First();
```

بتنفيذ الكود تصبح قيمة المتغير first تساوى 1.

FirstOrDefault

يقوم باسترجاع أول قيمة فى مجموعة عناصر. مع احتمالية ان تكون المجموعة فارغة.

```
int[] ints1 = { 1, 2, 3, 4, 5, 6 };
int[] ints2 = { } ;
Doctors doctors = new Doctors();
var query = from ...;

int x1 = ints.FirstOrDefault();
int x2 = ints.FirstOrDefault();
Doctor doctor = doctors.FirstOrDefault ( doc => doc.City == "Cairo" );
var result = query.FirstOrDefault ();
```

بتنفيذ الكود تصبح قيمة x1 تساوى 1 وقيمة x2 تساوى 0, لأن المجموعة ints2 فارغة.

Fold

يعتبر هذا الـ Operator مشابه لـ Aggregate Operator

GroupBy

بقوم هذا الـ Operator بتجميع العناصر الموجودة داخل مجموعة وفقاً لمفتاح (Key) والمفتاح هذا عبارة عن احد الجزاء المكونة للعنصر نفسه.

```
Doctors doctors = new Doctors();
var groups = doctors.GroupBy( doc => doc.City );
foreach ( var group in groups )
{
      Console.WritLine(" {0}: ", group.Key );
      Foreach ( var doc in group)
            Console.WriteLine("   {0}", doc.Name );
}
```

المتغير groups يحتوى على العناصر الموجودة داخل doctors مرتبة وفقاً لمفتاح وهو City. والمتغير group عبارة عن عنصر واحد فقط داخل groups. لذلك عند كتابة group.Key فذلك معناه قيمة المفتاح الذى رتبت بواسطتة القيم.

مثال أخر:

```
var groups = doctors.GroupBy( doc => doc.City );
var groups = from doc in doctors
            group doc by doc.City into g
            delect g;
var groups2 = doctors.GroupBy( doc => doc.City, doc => doc.Name );
var groups2 = from doc in doctors
            group doc.Name by doc.City into g
            select g;
```

GroupJoin

ربط مجموعتين ببعضهما عن طريق مفاتيح مختارة من المجموعتين. ثم عمل Group للبيانات بداخلهم.

```
DataSets.SchedulingDocs ds = FillDataSet();

var working = ds.Doctors.GroupJoin( ds.Calls,
                                    doc => doc.Initials,
                                    call => call.Initials,
                                    (doc, call) => new { doc.Initials,
                                     calls = cal }
);
```

```
foreach ( var record in working )
{
      Console.WriteLine( "{0}: ", record.Initials );
      foreach ( var call in record.Calls )
            Console.WriteLine( "   {0}", call.DaeOfCall );
}
```

Intersect

يقوم باسترجاع العناصر المتشابهة فى مجموعتين. بشرط أن تكون هذه العناصر غير متكررة فى مجموعتها.

```
int[] intsS1 = { 1, 2, 2, 3, 2, 3, 4, 5, 6, 8 };
int[] intsS2 = { 6, 1, 3, 6, 7 };

var query1 = from ...;
var query2 = from ...;

var commonInts = intsS1.Intersect(intsS2);
var commonResults = query1.intersect(query2);
```

فى تلك الحالة يمكننا أن نقول ان قيمة المتغير commonInts تساوى (1, 3, 6).

Join

ربط مجموعتين معاً من خلال مفتاح. مثلها مثل عبارة inner join فى SQL.

```
DataSets.SchedulingDocs ds = FillDataSet();

var working = ds.Doctors. Join( ds.Calls,
                              doc => doc.Initials,
                              call => call.Initials,
                              (doc, call) => new { doc.Initials,
                               calls = cal }
);
```

Last

يقوم باسترجاع أخر عنصر فى المجموعة.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
int last = ints.Last();
```

قيمة المتغير last تصبح ؟ .... (أكمل)

LastOrDefault

يقوم باسترجاع أخر عنصر من مجموعة يحتمل ان يكون بها قيم فارغة.

```
int[] ints1 = { 1, 2, 3, 4, 5, 6 };
int[] ints2 = { };
int x1 = ints1.LastOrDefault();
int x2 = ints2.LastOrDefault();
```

قيمة المتغير x1 تساوى 6. وقيمة المتغير x2 تساوى 0.

LongCount

تقوم بعد عناصر مجموعة محددة واسترجاع الناتج فى متغير من النوع Long

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
decimal?[] values = { 1, null, 2, null, 3 };

long count1 = ints.LongCount();
long count2 = values.LongCount();
```

قيمة count1 تساوى 6. و قيمة values تساوى 5.

Max

ايجاد أكبر قيمة لعنصر فى مجموعة.

```
int[] ints = { 1, 2, 3, 6, 5, 4 };
decimal?[] values = { 1, null, 4, null, 3, 2 };
var query = from ...;
int max1 = ints.Max();
decimal? Max2 = values.Max();
var max = query.Max();
```

قيمة المتغير max1 تساوى 6. وقيمة المتغير max2 تساوى 4.

Min

إيجاد أصغر قيمة فى مجموعة.

```
int[] ints { 6, 2, 3, 1, 5, 4 };
```

```
int min1 = ints.Min();
```

OfType

يقوم باستخراج العناصر الموجودة فى مجموعة بشرط أن تكون من نوع محدد.

```
System.Collections.ArrayList al = new System.Colliction.ArrayList();
al.Add(1);
al.Add("abc");
al.Add(2);
al.Add("def");
al.Add(3);
var strings = al.OfType<string>();
```

فى هذه الحالة تصبح قيمة strings تساوى (abc, def).

OrderBy

ترتيب عناصر المجموعة ترتيباً تصاعدياً وفقاً لأحد قيم العناصر. إذا كان لدينا مجموعة تتكون من بيانات أطباء. وكانت بيانات الأطباء تتكون من رقم الطبيب و أسمه و عنوانه و هاتفه ..الخ. يمكننا عرض بيانات الأطباء مرتبه وفقاً لأى قيمة من بيانات الأطباء مثلاً وفقاً لرقم الطبيب.

```
Doctors doctors = new Doctors();
Var doc = doctors.OrderBy(doc => doc.ID );
```

OrderByDescending

يقوم بنفس عمل الـ Operator السابق لكن هذه المرة يرتب البيانات ترتيباً تنازلياً.

Range

يقوم بتوليد سلسة من الأرقام الصحيحة (integers) محصورة بين رقمين يتم تحديدهم.

```
var oneToten = System.Query.Sequence.Range(1, 10);
```

Repeat

تكرار قيمة معينة لعدد معين من المرات. الرقم الأول هو الرقم المراد تكراره والثانى هو عدد التكرارات

```
var zeros = System.Query.Sequence.Repeat(0, 8);
```

Reverse

يقوم بعكس ترتيب العناصر الموجودة فى اى مجموعة.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
var revInts = ints.Reverse();
```

تصبح نتيجة revInts تساوى (6, 5, 4, 3, 2, 1)

Select

يمكننا هذا الـ Operator من أختيار قيم بيانات محددة من عنصر فى مجموعة عناصر. مثلاً مجموعة الأطباء تحتوى على عناصر هى الأطباء و كل طبيب يحتوى على عدة بيانات. يمكننا اختيار هذه البيانات أو جزء منها.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
var sameInts = ints.Select(x => x);
var names = doctors.Select( d=> new { d. LastName, d. FirstName } );
```

فى المتغير sameInts قمنا باختيار كل العناصر الموجودة فى ints. اما المتغير names قمنا بأختيار الأسم الأول و أسم العائلة فقط من بيانات الأطباء.

SelectMany

يقوم باسترجاع بيانات من مجموعة من المجموعات. مثلاً لدينا مجموعة, نوع العناصر بها عبارة عن array فى تلك الحالة تسمى مجموعة من المجموعات لأن الـarray تعتبر مجموعة.

```
// تعريف مجموعة من مصفوفات
List<int[]> list = new List<int[]>();
int[] ints123 = { 1, 2, 3 };
int[] ints456 = { 4, 5, 6 };
// إضافة المصفوفات الى المجموعة
List.Add(ints123);
List.Add(ints456);
var flat = List.SelectMany( x => x );
```

او استعلمنا عن العناصر فى List لكانت النتيجة (int[], int[]) اى مصفوفتين من النوع int. لكن مع استخدام selectMany مع المتغير flat, اصبح قيمته تساوى (1, 2, 3, 4, 5, 6).

Single

يقوم باسترجاع عنصر واحد فقط من المجموعة. شرط أن تكون المجموعة بها عنصر واحد فقط.

```
int[] ints = { 3 };
Doctors doctors = new Doctors();
Var query = from ...;

Int lone = ints.Single();
Doctor doctor = doctors.Single( doc => doc.Initials == "mb1" );
Var result = query.Single();
int pagerNum = doctor.PagerNumber;
```

بالنسبة للمتغير doctor نجد انه يحمل عنصر واحد فقط من المجموعة doctors وهو العنصر الذى قيمة Initials فيه تساوى mb1. هذا المتغير الأن يحمل العنصر كامل بكل بياناته.

SingleOrDefault

نفس الـ Operator السابق مع احتمالية أن تكون المجموعة فارغة.

Skip

يقوم هذا الـ Operator بتخطى عد معين من العناصر حسب ترتيبهم.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
var query = from ...;
var last3 = ints.Skip(3);
var bottom10 = query.Skip( query.Count() - 10 );
```

النتيجة... last3 يساوى (6 ,5 ,4) وذلك لأنه تخطى أول ثلاثة عناصر. و bottom10 سوف يقوم بعد جميع العناصر و يطرح منهم 10 بذلك سوف يسترجع جميع العناصر عدى أخر عشرة.

SkipWhile

أى تخطى العنصر الحالى إذا تحقق شرط معين.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
var Last3 = ints.SkipWhile( x => x < 3 )
```

قيمة المتغير Last3 تصبح (4, 5, 6).

Sum

يقوم بجمع قيم مجموعة ما.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
Doctors doctors = new Doctors();
int sum1 = ints.Sum();
double sumYears = doctors.Sum( doc =>
                DateTime.Now.Subtract( doc.StartDate).Days / 365.25 );
```

Take

يقوم باسترجاع كمية عناصر محدة ابتداءاً من أول عنصر. مثلاً إذا كان هناك مجموعة تحتوى على 40 عنصر باستخدام Take يمكننا مثلاً استرجاع أول 10 عناصر فقط مع اهمال الباقى. إذا كان الرقم المحدد لإسترجاعه أقل من 0 فإن النتيجه سوف تكون فارغة. وإذا كان أكبر من 0 فان النتيجة سوف تكون المجموعة كاملة.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
var query = from ...;
var first = ints.Take(3);
var top10 = query.Take(10);
```

يمكنكم استنتاج قيمة المتغير first.

TakeWhile

استرجع العناصر التى تحقق شرط محدد. هل تتذكر SkipWhile Operator ؟ لقد كان يترك العنصر إذا ما تحقق شرط معين. اما TakeWhile فهو العكس يأخذ العنصر إذا ما تحقق شرط معين.

ThenBy

انظ لهذا الكود أولاً:

```
Doctors doctors = New Doctors();
var docs = doctors.OrderBy(doc => doc.City).ThenBy(doc => doc.Name );
```

كما تلاحظ يتم استخدام GroupBy لترتيب العناصر وفقاً لقيمة City, ثم بعد ترتيب البيانات يتم ترتيبها مرة اخرى وفقاً للأسم. فلنفترض انه فى الترتيب الأول وهو الترتيب القائمة على City ان اول عشرة عناصر

لهم نفس قيمة City, يأتى ThenBy على هذه العناصر العشرة ثم يعيد ترتيبهم وفقاً لـ Name. وهكذا فى كل مجموعة عناصر لها City متشابهة.

ThenByDesecnding

نفس الـ operator السابق مع الترتيب تنازلياً.

ToArray

يقوم باسترجاع البيانات و تخزينها فى كائن array.

```
Doctors doctors = new Doctors();
var query = from doc in doctors
            where doc.City == "Cairo"
            select doc;
Docotrs[] Cairo = query.ToArray();
```

ToDictionary

استرجاع البيانات و تخزينها فى كائن يشبه المصفوفة ذات البعدين (two Dimension array). كل صف فى هذه المصفوفة لابد أن يحمل قيمة مختلفة عن باقى الصفوف. هذا الكائن يسمى<Dictionary<K, V.

```
Doctors doctors = new Doctors();
var query = from doc in doctors
            where doc.City == "Cairo"
            select doc;
Dictionary<string, Doctor> Cairo = query.ToDictionary( doc => doc.Initials);
```

الكائن Dictionary يأخذ قيمتين فى كل صف, القيمة الأولى تمثل المفتاح و الثانية تمثل القيمة المتعلقة بالمفتاح. فى الكود السابق انشانا كائن dictionary القيمة الأولى له هى doc.Initials و القيمة الثانية هى عنصر doctors كامل. ويمكن تحديد العنصر الثانى كى يصبح بيان واحد فقط و ليس صف بيانات. فبدلاً من أن يصبح عنصر doctors بكامل بياناته. يمكننا اختيار بيان واحد فقط.

ToList

استرجاع البيانات و تخزينها فى كائن <List<T.

List<Doctor> Cairo – query.ToList();

ToLookup

استرجاع البيانات و تخزينها فى كائن Lookup<K, V>. هذا الكائن يسبه كائن Dictionary<K, V> لكن ليس بالضرورى أن تكون القيم المخزنة قيم منفردة.

ToSequence

لشرح طريقة عمل هذا الـ Operator يجب أن تنظر الى الكود التالى أولاً:

```
Doctors doctors = new Doctors();
int count = doctors.Count( doc => doc.City == "Cairo" );
```

فى هذا الكود واضح إننا نريد عدد الطباء الذىين يعيشون فى Cairo. لكن هذا الكود سوف يتسبب فى خطأ عند تنفيذ الكود. وذلك لأن المجموعة doctors لها خاصية تسمى count وايضاً الـ operator الخاص بـLinq يسمى count لذلك سوف يحدث تعارض بين الخاصية و الـ operator. ولن ينفذ الكود, لذلك يظهر ToSequence كحل سريع لهذه المشكلة.

```
int count = doctors.ToSequence().Count( doc => doc.City == "Cairo" );
```

Union

يقوم باسترجاع البيانات المتشابهة فى مجموعتين. او بصيغة أخرى يقوم باسترجاع اتحاد فئتين.

```
int[] intS1 = { 1, 2, 2, 3, 2, 3, 4, 6 };
int[] intS2 = { 6, 1, 3, 5 };
var allInts = intS1.Union(intS2);
```

تصبح قيمة المتغير allInts تساوى (1, 2, 3, 4, 6, 5). وكما نلاحظ انه استرجع بيانات الفئة الأولى اولاً ثم الفئة الثانية.

Where

استرجع البيانات التى تحقق الشرط.

```
int[] ints = { 1, 2, 3, 4, 5, 6 };
var even = ints.Where(x => x % 2 == 0);
```

3

الفصل الثالث

LINQ To SQL

نتعرف فى هذا الفصل على LINQ to SQL

LINQ to SQL

أول و أهم جزء فى LINQ هو الجزء الخاص بالإستعلام من البيانات العلائقية أو قواعد البيانات المتعارف عليها. هذا الجزء فى LINQ يقدم لك طريق سهلة للتعامل مع البيانات المخزنة فى قواعد البيانات. حيث تقوم LINQ بتحويل ما كتبته من كود الى استعلام SQL و ترسله الى قاعدة البيانات. انظر المثال التالى:

```
var query = from c in Customers
      where c.Country == " USA"
            && c.State == "WA"
      Select new { c.CustomerID, c.CompanyName, c.City };
```

هذا الكود سوف يتحول الى استعلام SQL بهذا الشكل:

```
SELECT CustomerID, CompanyName, City
FROM Customers
WHERE Country ='USA' AND Region = 'WA'
```

الى هذا الحد ربما تريد أن تسأل عدة اسئلة. اولاً كيف يمكن لإستعلام LINQ ان يكتب باستخدام اسم الكائن و يتم التحقق من صحته بواسطة المترجم؟ فى الماضى كان التحقق من الإستعلام يتم على يد DBMS و ليس مترجم اللغة. ثانياً متى يتم توليد استعلام SQL من الإستعلام الذى نكتبه باستخدام LINQ؟. ثالثاً متى يتم تنفيذ استعلام SQL؟. حسناً من حقك علينا أن نقوم بإجابة تلك الأسئلة. لكن لمعرفة الإجابات يجب عليك أن تفهم ما هو نموذج الكينونة (Entity Model) الخاص بـ LINQ to SQL و كذلك يجب أن تفهم ما هو الإستعلام المؤجل (Deferred Query).

الكينونات فى LINQ to SQL

اى بيانات خارجية (اى خارج نطاق الكود) لابد أن يتم وصفها وصفاً تفصيلياً بداخل الكود. يجب أن يكون هناك فئة (Class) لأى جدول, وتلك الفئة لابد أن يكون لها Attributes تصف صف البيانات الموجود بداخل الجدول. اى تصف كل الحقول الموجودة فى الجدول.طبعاً من خلال دراستك للغة C# تعرضت للـ Attributes وماهى و كيف يتم التعامل معها. لذلك لن نتحدث عنها هنا. الكود التالى يمثل تعريف لكينونة:

```
[Table (Name = "Customer")]
public class Customer
{
      [Column] public string CustomerID;
      [Column] public string CompanyName;
      [Column] public string City;
      [Column(Name = "Region"] public string State;
      [Column] public string Country;
}
```

الـ Attributes تلك التى مكتوبة بين القوسين [ ]. بهذا الشكل تلك الفئة أصبحت نموذج للجدول أو نموذج للكينونة.

لحسن الحظ انك لست مضطر لتعريف نموذج الكينونة لكل كينونة تستخدمها. ولا حتى لأى كينونة. فعن طريق استخدام Visual Studio يمكنك اضافة ملف dbml و اختيار قاعدة البيانات و الجداول التى سوف تتعامل معها ويقوم Visual Studio بتوليد الكود اللازم بدلاً منك وكل ما عليك فعلة لإستخدام هذا الكود هو عمل نسخة (Instance) من الملف واستخدامه كالتلى:

```
DataContext db = new DataContext();
```

كما ترى فإن اسم الملف الذى ولده Visual Studio هو DataContext فى الحقيقة الملف اسمه Data فقط وهذا الأسم من اختيارك انت اما كلمة Context فهى كلمة يضيفها Visual Studio.

فى هذه الحالة و بعد عمل نسخة من ملف dbml يمكننا الأن كتابة الإستعلام المناسب:

```
var query = from c in  db.Customer
      where c.Country == "USA"
            && c.State == "WA"
      Select new { c.CustomerID, c.CompanyName, C.City };

foreach( var row in query )
{
      Console.WriteLine( row );
}
```

فى البادية قمنا بتعريف متغير من النوع var اى انه غير محدد النوع (Anonymous Type). وهذا المتغير يعبر عن نتيجة تنفيذ إستعلام. وهذا الإستعلام تفصيله كالتالى :

من c الموجود فى جدول Customer

عندما country = USA و State = WA

اختار CustomerID, CompanyName, City.

المتغير query الأن اصبح يحمل النتيجة المسترجعة من تنفيذ هذا الإستعلام. ثم بعد ذلك تقوم جملة foreach بالدوران بداخل المتغير الحاوى لنتيجة الإستعلام وتقوم بعرض كل سطر على الشاشة.

Stored Procedures

استخدام Stored Procedure لإسترجاع البيانات يعتبر من الأمور المهمة فى عالم قواعد البيانات. لذلك إذا كان لديك قاعدة بيانات تحوتى على Stored Procedures فيمكنك استدعائهم بواسطة LINQ بمنتهى السهولة. لكن, يجب تعريف دالة تحتوى على وصف للـ Procedure المطلوب تنفيذه وطبعاً تلك الدالة يجب أن يكون لها Attributes هى التى تصف الـ Procedure كما فعلنا سابقاً مع الجداول. واحب أن

اقول لك لا تقلق فلن تقوم بتعريف أو وصف اى شىء بنفسك. فتعريف تلك الدالة يتم داخل ملف dbml ايضاً. نفس الملف الذى استخدمته لتعريف الجداول. لكن لا ضرر من النظر الى شكل الدالة:

```
[SoredProcedure( Name = "dbo.[Customer By City]")]
public IEnumerable<CustomerInfo> CustomerByCity(String parm1)
{
      Return (IEnumerable<CustomerInfo>)
            this. ExecuteMethodCall<CustomerInfo>(
            this, ((MethodInf) (MethodInfo.GetCurrentMethod())), parm1);
}
```

اعتقد ان الكود لا يحتاج الى شرح.... ليس لبساطتة بالطبع فهو معقد من الدرجة الأولى. لكن لأننا لن نكتبه. فى تلك الحالة و بعد أن تم تعريف الـ procedure على انه دالة بداخل الكود يمكننا استدعاؤه كالتالى:

```
var query = db.CustomerByCity("Cairo");
```

Compiled Query

احياناً تحتاج الى إعادة استخدام نفس الإستعلام مع قيم مختلفة... فى الحقيقة غالباً ما تحتاج الى هذا... بل انى اكاد أجزم انه دائماً ما تحتاج الى هذا. لذلك, تقوم بعض الـ DBMS بعمل Optimization (تحسين) للإستعلام المستقبل من اى تطبيق (Application) وذلك لتحسين الأداء و عملية ترجمة (compilation) الإستعلام. ذلك ليؤدى الى زيادة أداء التطبيق الذى ارسل الإستعلام نفسه, لأن الـ DBMS لن يضيع الوقت فى أعادة تحليل الإستعلام فى كل مرة يتم ارساله فيها. تقدم LINQ استراتيجية لعمل Optimization للإستعلام, لكن فى كل مرة تقوم بارسال الإستعلام, يقوم محرك LINQ بتوليد استعلام مناسب بلغة SQL لإرساله الى DBMS. لذلك تقدم لك LINQ طريقة جيدة لعمل Optimization وذلك عن طريق استخدام فئة تسمى CompiledQuery. باستخدام تلك الفئة لن يحتاج المترجم الى ترجمة الإستعلام فى كل مرة تطلبه فيها. انظر الكود التالى:

```
DataContext db = new DataContext();
Table<Customer> Customers = db.GetTable<Customer>();

var query = CompiledQuery.Compile(
            ( DataContext context, string filterCountry ) =>
            from c in Customers
            where c.Country == filterCountry
            select new { c.CustomerID, c.CompanyName, c.City } );

foreach ( var row in query(db, "USA"))
{
      Console.WriteLine( row );
}

Foreach ( var row in query(db, "Italy"))
{
      Console.WriteLine( row );
}
```

كما ترى, استخدمنا دالة ()CompiledQuery.Compile وكتبنا الإستعلام كأنه عنصر فيها. اما الإستعلام نفسه فقط كتب على شكل Lambda Expression. وقمنا باستدعاء الإستعلام مرتين كل مرة بقيمة مختلفة عن الأخرى.

كما هو ملاحظ أن نتيجة تنفيذ الإستعلام سوف تخزن فى كائن var. حسناً, ماذا لو اردت يوماً ما أن تخزن نتيجة الإستعلام فى كائن static لسهولة إعادة الإستخدام.... اعتقد انه فى تلك الحالة يجب عليك ان تنظر الى الكود التالى:

```
public static Func<nwind.Northwinf, string , IQueryable<nwind.Customer>>
      CustomerByCountry =
            CompiledQuery.Compile(
                  ( nwind.Northwind db, string filterCountry ) =>
                        from c in db.Customers
                        where c.Country == filterCountry
                        select c );
static void CompiledQueriesStatic()
{
      nwind.Northwind db = new nwind.Northwind( ConnectionString);

      foreach ( var row in CustomerByCountry( db, "USA" ))
      {
            Console.WriteLine(row.CustomerID);
      }
      foreach ( var row in CustomerByCountry( db, "USA" ))
      {
            Console.WriteLine(row.CustomerID);
      }
```

بعد رؤية هذا الكود أعتقد انك تفكر فى عدم استخدام هذا الذى يسمى Compiled Query ولا داعى لعمل اى Optimization لأى شىء. لكن فى الحقيقة الكود ليس بصعوبة شكله. لكن إن فهمت الكود جيداً فلن تجد اى صعوبة فى عمله بل بالعكس قد لا تستخدم فى برامجك سوى Compiled Query. المطلوب منك فقط هو معرفة لغة #C و التحسينات الجديدة التى طرأت عليها. و هناك العديد من الكتب التى تتحدث عنها.

طرق مختلفة للإستعلام عن البيانات

الإستعلام التالى مكتوب بلغة SQL وهو يقوم بحساب مجموعة كمية من منتج تم بيعه. اى استعلام لمعرفة كم وحدة من منتج معين تم بيعها. المنتج فى هذا الإستعلام اسمه Chocolade.

```
SELECT SUM( od.Quantity ) AS TotalQuantity
FROM [Prosucts] p
Left Join [Order Details] od ON
           od.[ProductID] = p.[ProductID]
WHERE p.ProductName = 'Chocolade'
Group By p.ProductName
```

لو كتبنا ذلك الإستعلام باستخدام LINQ فسوف يبدو كالتالى:

```
var queryJoin =
      from p in db.Products
      join o in db.Order_Details
            on p.ProductID equals o.ProductID
            into OrdersProduct
      where p.ProductName == "Chocolade"
      select OrdersProduct.Sum( o => o.Quantity );

var quantityJoin = queryJoin.Single();
Console.WriteLine( quantityJoin );
```

كما هو ملاحظ اننا استخدما Join لتحديد العلاقة بين الجدولين تحديداً صريحاً لا يشوبه اى خطأ. تسمح لنا LINQ بكتابة استعلام دون تحديد صريح للعلاقة التى بين الجدولين. كالتالى:

```
var queryAssociation =
      from p in db.Products
      where p.ProductName == "Chocolade"
      select p.Order_Details.Sum(o => o.Quantity);
var quantityAssociation = queryAssociation.Single();
Console.WriteLine( quantityAssociation );
```

عندما تقوم LINQ بتحويل الإستعلامين السابقين الى كود SQL فهى تنتج نفس الكود فى الحالتين. كل الفرق بينهما أن الكود الأول و الذى تم تحديد العلاقة فيه تحديداً صريحاً أكثر فهما من الكود الثانى لكنه أطول واحتمالية الخطأ فى كتابة العلاقة موجودة. فى الحالتين لن تهم LINQ اى طريقة تستخدم . بل انت الذى قد يهتم بأى طريق سوف تكتب الكود الخاص بك... الأمر يعود اليك.

هناك شيء أخر قد تحب معرفته. بالنسبة للإستعلام السابق, ليس هناك داعى لكتابة استعلام كامل, كل ما تريده من هذا الإستعلام هو قيمة واحدة فقط و هى كمية المنتج التى بيعت. يمكنك كتابة الكود التالى باستخدام بعض الـ operators السابق شرحها يمكنك معالجة الأمر بطريقة سريعة:

```
var chocolade = db.Products.Single( p =>.ProductName = "Chocolade" );
var quantityValue = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityValue );
```

الخطوة الأولى فى الكود السابق تقوم بتحديد اليكنونة التى نتعامل معها. و الخطوة الثانية تقوم بالدخول الى جدول Order_Details لحساب الكمية. من النظرة الأولى يترائى لك أن هذا الكود أقصر فى كتابته مقارنة بالإستعلام الكامل, لكنه للأسف أدائه اسوء من اداء الإستعلام ... قد يكون هذا الكود صحيح و مناسب إذا كنت تريد إجراء عملية واحدة فقط مع إهمال باقى الظروف. لكنه قد يكون غير مناسب فى الإستخدام العام.

تضمن لك LINQ تخزين نسخة من الكينونة التى تتعامل معها فى الذاكرة. إذا اردت اجراء اى استعلام اخر أو قيمة أخر لنفس الإستعلام على نفس الكينونة فهى موجودة فى الذاكرة بالفعل. الكود السابق المختصر لم يقم بعمل نسخمة من الكينونة فى الذاكرة لأنه استرجع بيان واحد فقط. من هذه النقطة, لو ان هذا الكود قام بعمل نسخة من الكينونة داخل الذاكرة فإن اداء الإستعلام سوف يكون فى منتهى السوء لأنه و بمنتهى البساطة قام بعمل نسخة من كينونة و علاقة بينها و بين جدول أخر فقط ليحسب مرة واحدة فقط كمية المنتج Chocolade الذى تم بيعه.

قد تعتقد إننا قمنا بعمل إستعلامين عندما دخلنا على البيانات باستخدام الكينونة Product لأننا استخدمنا جملة لتخصيص المتغير choclade و جملة أخرى لحساب الكمية. هذا الإفتراض غير صحيح تماماً. حتى لو كتبنا جملة واحدة فقط , فإن استخدام الكينونة Product سوف ينتج نفس النتيجة (الخاصة بالكائنات التى فى الذاكرة و معالجة SQL).

```
var quantityChocolade = db.Products.Single( p => p.ProductName == "Change" )
      .Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityChocolate );
```

يبدو أبسط ... الكود السابق بالرغم من انه جملة واحدة إلا انه نتيجة تنفيذه (الخاصة بالأداء وليس بالعائد) متشابهة من الكود الذى قبله. فى الحقيقة إيجاد طريقة مناسبة للدخول الى البيانات تعتمد على جميع العمليات و الإستعلامات التى يقوم بها البرنامج كله. فالوصول الى البيانات عن طريق الدخول الى الكينونات ربما يقدم أداء أفضل. لكن فى الجانب الأخر إذا كنت تحصل على نتيجة الإستعلام باستخدام أنواع مجهولة (Anonymous types) ولا تقوم بالعمل على الكينونات فى الذاكرة, فربما تفضل أن تعمل بطرق تعتمد على الإستعلامات.... بناءاً على تلك الحالات السابقة يمكننا أن نقول إن الطريقة الأمثل للدخول الى البيانات " يعتمد على ".

الإستعلامات المباشرة (Direct Queries)

فى بعض الأحيان قد تحتاج الى استخدام بعض ميزات SQL الغير موجودة فى LINQ. على سبيل المثال مقد تحتاج الى استخدام ميزة Common Table Expressions (CTE) أو استخدام أمر PIVOT. لا تملك LINQ بنية واضحة لهذه الميزات. المثال التالى يوضح لك يكفية استخدام دالة ExecuteQuet<T> لإرسال استعلام مباشر لقاعدة البيانات. بخصوص حرف T الموجود فى ExecuteQuery<T> فهو يمثل الكينونة التى سوف نستخرج منها البيانات. بخصوص الكود التالى, ليس مطلوب منك معرفة ماذا يفعل, بل كل ما هو مطلوب أن تعلم أنك يمكنك كتابة استعلام SQL مباشرة بداخل LINQ.

```
var query = db.ExecuteQuery<EmployeeInfo>(@"
      With EmployeeHierarchy (EmployeeID, LastName, FirstName,
            ReportsTo, HierachyLevel) AS
      ( SELECT EmployeeID, LastName, ForstName,
            ReportsTo, 1 as HierarchyLevel
      FROM Employees
      WHERE ReportsTo IS NULL

      UNION ALL

      SELECT e.EmployeeID, e.LastName, e.FirstName,
            e.ReportsTo, eh.HierarchyLevel + 1 AS HierarchyLevel
      FROM Emplyees e
      INNER JOIN EmployeeHierarchy eh
            ON e.ReportsTo = eh.EmplyeeID
      )
      SELECT *
      FROM EmployeeHierarchy
      ORDER BY HierarvhyLevel, LastName, FirstName" );
```

Read-Only DataContext Access

إذا كنت ترغب فى الدخول الى البيانات للقراءة فقط بدون إمكانيات التعديل فيها, فيجب عليك أن تقوم بإيقاف الخدمة التى تدعم تعديل البيانات والتى توجد بداخل الـDataContext.

```
DataContext db = new DataContext( ConnectionString );
Db.ObjectTracjing = false;
```

تحديث البيانات (Data Update)

هناك خدمة (Service) بداخل LINQ To SQL تسمى identity management, وهى تقوم بتعقب الكينونات التى تتعامل معها, هذه الخدمة مضمونة فقط للكينونات التى تأتى عن طريق DataContext. فهى تضع كل صف موجود فى الكينونة فى الذاكرة .

تحديث الكينونات

هناك خدمة فى LINQ تسمى Change tracking اى تتبع التغيرات. تلك الخدمة مسؤلة عن تغيرات البيانات التى تحدث فى الكينونات. حيث تحتفظ تلك الخدمة بالقيم الأصلية للبيانات و ايضاً القيم الجديدة, وتقوم بتوليد استعلام SQL لتحديث البيانات فى قاعدة البيانات نفسها. ويمكنك ان ترى كود SQL الذى ولدته تلك الخدمة عن طريق استخدام دالة GetChangeText.

```
var customer = db.Customers.Single( c => c.CustomerID == "FRANS" );
customer.ContactName = "Marco Russo";
Console.WriteLine( db.GetChangeText() );
```

الكود السابق قام بتغير قيمة Contact Name الى Marco Russo عند العنصر الذى يكون CustomerID فيه يساوى FRANS. كود SQL الذى يتم توليده من هذا الكود كالتالى:

```
UPDATE [Customers]
SET [ContactName] = 'Marco Russo'
FROME [Customers]
WHERE ...
```

تذكر جيداً أن ذلك كود SQL تم توليده فقط, اى انه لم يرسل بعد الى قاعدة البيانات. فهو لن يرسل إلا بعد استدعاء دالة SubmitChanges.

اما إذا اردت أن تضيف سجل للجدول أو تحذف سجل من الجدول, فإن إنشاء السجل و وضعه فى الذاكرة ليس كافياً. فيجب بعد إنشاء السجل فى الذاكرة أن نضيفه الى قاعدة البيانات. النظر الكود التالى:

```
var NewCustomer = new customer {
                              CustomerID = "DLEAP",
                              CompanyName = "DevLeap",
                              Country = "Italy" };
db.Customers.Add( newCustomer );

var oldCustomer = db.Customers.Single( c => c.CustomerID == "FRANS" );
db.Customer.Remove( oldCustomer );
```

كما هو وضاح قمنا بعمل نسخة أو سجل من الكائن customer و اسميناه newCustomer و اعطيناه القيم المناسبة. ثم بعد ذلك استخدمنا دالة Add لإضافة السجل فى الجدول. ثم قمنا بالبحث عن السجل الذى قيمة CustomerID فيه تساوى FRANS. ثم باستخدام دالة Remove قمنا بمسحه من الجدول... لا يوجد أسهل من ذلك... انظر الى كود SQL الذى يتم توليده خلف الكواليس:

```
INSERT INTO [Customers](CustomerID, CompanyName, ...)
VALUES("DEVLEAP", "DevLeap", ...)

UPDATE [Orders]
SET    [CustomerID] = NULL
FROM   [Orders]
WHERE  ([OrderID] = @p1) AND ...

DELETE FROM [Customers] WHERE [CustomerID] = "FRANS"
```

أعتقد انك تعرف كود SQL جيداً و تدرك معانيه... اى انك لا تحتاج لشرح هذا الكود... ولو افترضنا فرضاً خيالياً انك لا تدرك معانيه, فلا تدع هذا يصيبك بالقلق, لقد وضعنا الكود هنا لعرضه فقط.

بخصوص حذف السجلات من الجداول يمكن استخدام Remove او RemoveAll فى المثال الاسبق استعملنا Remove و فى المثال التالى سوف نستعمل RemoveAll.

```
var order = db.Orders.Single( o => o.OrderID == 10248 );
db.Order_Details.RemoveAll( order.Order_Details );
db.Order.Remove( Order );
db.SubmitChanges()
```

تحديث قواعد البيانات

عند استخدام LINQ To SQL, فإن العديد من جمل SQL يتم توليدها و ارسالها الى قاعدة البيانات بطريقة غير مرئية بالنسبة للمستخدم. على الجانب الأخر فإن اوامر SQL التى تقوم بتعديل حالة البيانات فى القاعدة ترسل الى القاعدة عنا تقرر انت ذلك عن طريق استخدام دالة SubmitChanges الموجودة فى DataContext. كما هو موضح فى الكود التالى :

```
Northwind db = new Northwind( Program.ConnictionString );
var customer = db.Customers.Single( c => c.CustomerID == "FRANS";
customer.ContactName = "Marco Russo";
db.SubmitChanges();
```

العمليات المتزامنة (Concurrent Operations)

تقوم LINQ بتخزين الكينونات التى تتعامل معها فى الذاكرة. أى انها تعمل بنظام Disconnected. فى هذه الحالة, غالباً ما يحدث أن يقوم أكثر من مستخدم بإجراء عمليات متزامنة فى وقت واحد. وقد يتسبب هذا التزامن فى إجراء العمليات فى حدوث تعارض بين المستخدمين. فى تلك الحالة يصدر المترجم رسالة تفيد بأن هناك تعارض فى العمليات, هذه الرسالة تحتوى على العمليات المتعارضة وسبب الخطأ. ويتم إيقاف العمليات. الكائن المسؤل عن إصدار تلك الرسالة هو ChangeConflictExeption. ومن خلال دراستك للغة #C تعرض لدراسة الـ Exeptions. انظر الكود التالى:

```
Northwind db2 = new Northwind( Program.ConnictionString );
for ( int retry = 0; retry < 4; retry++ )
{
      var customer2 = db2.Customers.Single( c => c.CustomerID =="FRANS" );
      customer2.ContactName = "Paolo Pialorsi";
            try
            {
                  Db2.SubmitChanges();
                  Break;
            }
            Catch ( ChangeConflictExeption ex)
            {
                  Console.WriteLine( ex.Message );
                  Db2.Refresh( customer2, RefreshMode.KeepChanges );
            }
}
```

هذا الكود يقوم بإعادة محاولة تحديث البيانات حتى اربع محاولات, إذا وجد أى تعارض عند التنفيذ. وإذا تم التنفيذ بالفعل تتوقف المحاولات. و إن وجد أى تعارض, فإنه يصدر رسالة بمعلومات هذا التعارض.

طريقة أخرى لإعادة محاولة تحديث البيانات إذا حدث أى تعارض, فدالة SubmitChanges يمكن أن تحتوى على عناصر لتحديد إذا ما كنت تريد ايقاف التنفيذ أو إعادة المحاولة. لكن إن لم تحدد لها اى شىء فإن التصرف الإفتراضى لها هو أن تتوقف عن التنفيذ فى حالة التعارض.

```
Db.SubmitChanges(ConflictMode.FailOnFirstConflict);
Db.SubmitChanges(ConflictMode.ContinueOnConflict);
```

الأن, والأن فقط أصبحت مؤهلاً للبدأ فى استخدام LINQ. تلك التقنية التى سوف تصبح من اليوم أهم جزء فى برامجك. سواء كنت تعمل على قواعد بيانات أو تعمل على اى شىء أخر.

تم بحمد الله

11 - 1 - 2008

المراجع

- Introducing Microsoft LINQ, Microsoft Press, 2007
- LINQ - the future of data access in C# 3.0, O'Reilly Press, 2006
- msdn2.microsoft.com
- www.davidhayden.com